version 0.5
February 26th, 2004

L2 Processor Controls and Interfaces

Kristian Hahn, Joe Kroll, Chris Neu, Paul Keener,
Rick Van Berg, Peter Wittich, Daniel Whiteson
University of Pennsylvania

Abstract

A description of the architecture of the L2 processor nodes is given and the interfaces between the components is specified.

1 Introduction

2 Requirements

The L2 processing system must meet the following requirements with respect to monitoring and control:

2.1 L2 Processor Requirements

- monitoring should not be intrusive
- errors should be reported quickly

2.2 Run Control Requirements

The system should respond to the following commands from Run Control

- Partition
  The system will join the specified partition and listen only to commands from that partition until released. Acknowledgement or error report required.
- Configure
  The system will receive configuration information from RunControl, in the form of information from the Run, Trigger, or Hardware Databases. It will configure the nodes for event processing and respond with success or failure. Acknowledgement or error report required.
- Activate or Run
  The system will prepare for data-taking. Acknowledgement or error report required.
- Halt
  The system will halt processing events. Acknowledgement or error report required.
- Recover
  The system will flush events from the system and reset all queues. Acknowledgement or error report required.
- End
  The system will cease processing events. Acknowledgement or error report required.
- Reset
  The system is released from the specified partition. Acknowledgement or error report required.

2.3 Monitoring Requirements

The system will maintain and provide access to the following monitoring information, both on demand and on a regular basis.

For each node:

- CPU and memory usage for algorithm and IO
- Per trigger and total algorithm time
- Per trigger and total unpacking time
- Interrupt counts
- Ethernet statistics
- Disk usage
- Event size
- Number of trigger objects
- CPU and device IDs
- uptime

For the system:

- L1 Accept rate per trigger
- L2 Accept rate per trigger
- Number of filled buffers

3 System Architecture

3.1 T.E.D.

The architecture of the system is depicted in Figure 1; the individual processing nodes are isolated from the interface to the data acquisition system and from monitoring infrastructure by the Trigger Evaluator Director (TED). The functions of TED are separated into pieces which divide the task into the overall control and interface to external components:

- **Run Control Client**: The RCC will communicate with Run Control, translating its requests into appropriate L2 actions, and indicating the status of the L2 nodes.
- **L2 Node Controller**: The L2NC will issue commands to the L2 nodes, including requests for configuration, changes to processing modes and requests for monitoring data. Ine L2NC will handle communication with exactly one node.
- **L2 Node Monitor**: The L2NM will recieve monitoring data from the L2 Nodes. One L2NM will handle communication with exactly one node.
- **L2 Monitor Server**: The L2MS will allow access via a Monitoring GUI to the gathered monitoring information.
- **Central Control**: The Central Controller will own the above interfaces, and determine what actions to take in response to commands from external components.

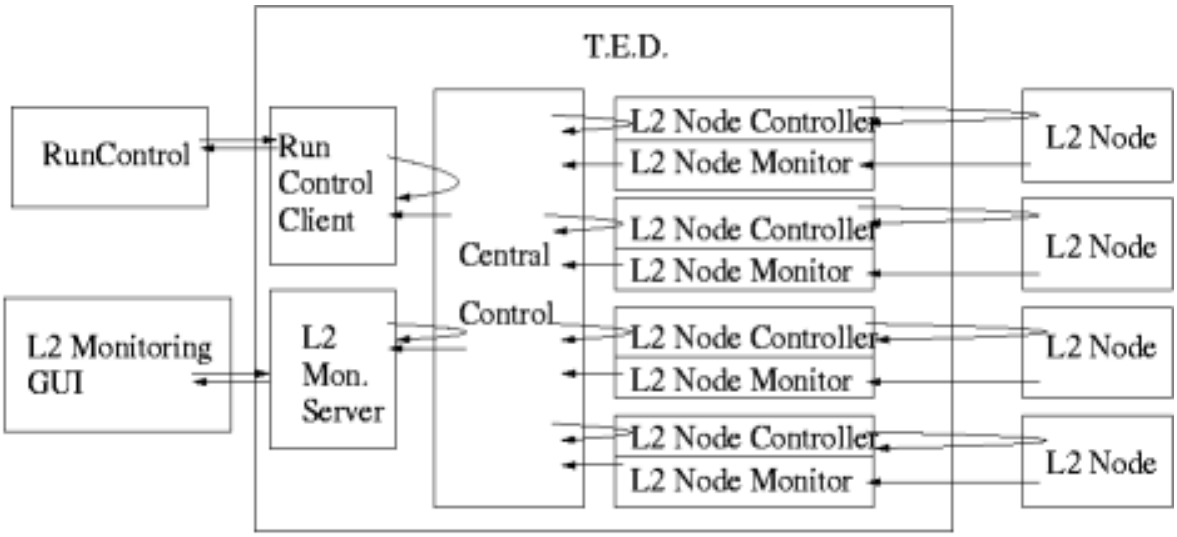

Figure 1: Communcation between pieces of the T.E.D, and their communication with outside components. Looped-back arrows indicate synchronous control which waits for an indication of success or failure; straight arrows indicate asynchronous data flow.

3.2 L2 Processors

Figure 2 depicts the division of tasks on the L2 processor nodes. Components shown

in the diagram are described below.

- **NodeAlgo :** This process deals with the evaluation of the trigger algorithms as well as S-LINK I/O.
- **NodeMonitor :** The NM gathers hardware information from the system and monitoring data from NodeAlgo, packages it and sends it to T.E.D.
- **NodeController :** The NC receives configuration, processing and monitoring commands from the L2NC on T.E.D. The interperted commands guide NC's control of NodeAlgo and NodeMonitor.

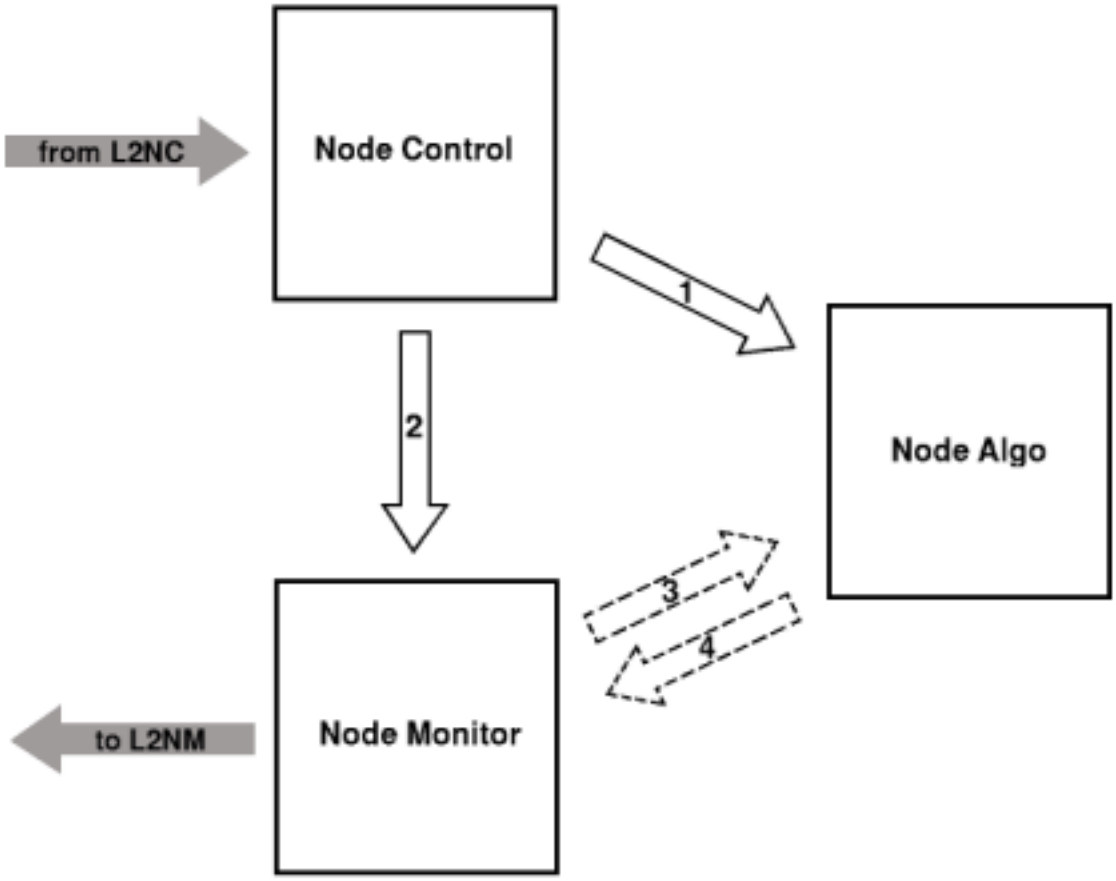

Figure 2: **L2 node Flow Control** This figure depicts the funcional blocks that operate on the L2 nodes. Solid and hashed bordered arrows represent synchronous and asynchronous operations, respectively. Filled grey arrows imply external communication with T.E.D. The single-headed arrows shown point from initiator to target but information, such as acknowlgements and requested data, will also flow in the counter direction.

4 T.E.D. Control Flow

T.E.D. is driven by its response to external asynchronous events; it acts as a central communication point, translating requests from one piece of the system into actions by other pieces.

The pieces of the system which interact with external components (Interface Components) operate asynchronously; external events may occur at any time and at any rate. The Central Control is a fairly simple object, which provides a set of routines which the Interface Components may call when an external event is detected. In this way, these routines provide an interface by which incoming messages could be translated into actions; it defines the response of the system to messages from external components.

4.1 Synchronous Control

RunControl Client

The RunControl Client listens for instructions from Run Control; when one arrives, the RCC is required to communicate its success or failure to RunControl. During the execution of the command, no further RC commands will be seen. Therefore, the RCC client may execute these commands synchronously, waiting to report their completion and status before listening for new commands. The RCC's interface to the Central Control will be as follows:

- `bool CentralControl::Configure(ConfigSpec);`
  Send configuration information to the nodes. For example, ConfigurationData may contain the version of the trigger table and the set of prescales. Return value indicates success or failure.

- `bool CentralControl::Activate();`
  Indicate to the nodes that they should be prepared for data to arrive. Return value indicates success or failure.

- `bool CentralControl::Run();`
  Prepare for data. Return value indicates success or failure.

- `bool CentralControl::Recover();`
  Flush the system of data and reset all queues. Return value indicates success or failure.

- `bool CentralControl::Halt();`
  Stop processing data. Return value indicates success or failure.

- `bool CentralControl::End();`
  The run has ended. Return value indicates success or failure.

L2 Node Controller

The L2 Node Controller issues instructions to the nodes; it waits to receive an acknowledgement from the nodes that the command has been successfully executed, though it does not retrieve any data nor wait for any further information. If data is sent to T.E.D. in response, it is passed via the L2 Node Monitor.

- `bool L2NodeController::Configure(ConfigData);`
  Instructs the node to configure; return result indicates success or failer
- `bool L2NodeController::GoThirsty();`
  Instructs the node to prepare for data; return result indicates success or failer
- `bool L2NodeController::GoDrunk();`
  Instructs the node to ignore further data; return result indicates success or failer
- `bool L2NodeController::Flush();`
  Instructs the node to discard data and clear queues; return result indicates success or failer
- `bool L2NodeController::RequestMonitoringData()`
  Instructs the node to send updated monitoring data at its next opportunity; return result indicats success or failure.

L2 Monitor Server

The L2 Monitor Server listens for requests for updated monitoring data from an external interface. It makes a request to the Central Control, waiting to hear an acknowledgement that the request was successfully transmitted. The monitoring data, however, is returned asynchronously, at the next available opportunity. The L2 Monitor Server makes use of the following interface to Central Control:

- `bool CentralControl::UpdateMonitoringData();`
  Request updated monitoring information from nodes at next opportunity.

4.2 Asynchronous Control

Monitoring and status information from the nodes passes to T.E.D. in an asynchronous manner; the nodes may report a processing error which requires resetting the system (issuing an HRR), or they may simply send monitoring statistics, a low priority task in comparison to data processing.

Central Control

The Central Control will pass monitoring data from the L2 Node Monitor to the L2 Monitoring Server when it arrives from the nodes:

- `bool CentralControl::StoreMonitoringData(NodeID, RunID, MonData);`
  Receive new monitoring data
- `bool L2MonitorServer::StoreMonitoringData(NodeID, RunID, MonData);`
  Receive new monitoring data

RunControl

If one of the nodes reports an error, seen via the L2MonitorServer, Central Control will indicate to the RCC to send an error to Run Control:

- `bool RunControlClient::L2Error(EventID, RunID);`
  Report an error

5 Node Control Flow

The evaluation of L2 trigger algorithms is the central process on the L2 nodes. The control and monitoring of this process is directed by T.E.D. through its L2 Node Controller and L2 Node Monitor interfaces. Corresponding interfaces exist on the L2 nodes (NodeControl & NodeMonitor) and establish a communication channel with T.E.D. External communication with T.E.D. and error condtitons on the nodes occur asynchronously. The flow of control initiated by these asynchronous events may be classifed as etiher synchronous or asynchronous. Sections 5.1 and 5.2 describe and categorize the interface functions used in the communication between processes on the nodes.

5.1 Synchronous Operations

NodeController

The synchronous operation of the L2NC on T.E.D. allows a node's NC to respond to single instructions from the L2NC before considering the next. The interface functions available to the NC mirror the commands used by the L2NC. Return values are sent back to the L2NC to indicate the sucessful or unsucessful completion of the command.

- **bool NodeAlgo::configureAlgo(AlgoConfigSpec), (1)**
  Configuration data, such as prescales and trigger table information, is passed to NodeAlgo.

- **bool NodeAlgo::setAlgoThirsty(), (1)**
  NodeAlgo is told to prepare for the arrival of data.

- **bool NodeAlgo::setAlgoDrunk(), (1)**
  NodeAlgo is told to ignore further data.

- **bool NodeAlgo::flush(), (1)**
  NodeAlgo is told to discard the current event and clear any queued information.

- **bool NodeMonitor::configureMonitoring(MonConfigSpec), (2)**
  Configuration data, such as sampling rate and ROI, is passed to NodeMonitor.

- **bool NodeMonitor::requestMonitoringData(), (2)**
  Request that monitoring data from NodeMonitor be sent right away.

5.2 Asynchronous Operations

NodeMonitor

One of NodeMonitor's tasks is to collect profiling information from NodeAlgo. This operation is asynchronous so that NodeMonitor can perform its other duties (fetching HW statistics, formatting data, etc.) while it waits for information from NodeAlgo to arrive.

- **bool NodeAlgo::getAlgoInfo(MonSpec), (3)**
  Request monitoring infomation from NodeAlgo. The information arrives when it becomes available.

NodeAlgo

NodeAlgo passes profiling data to NodeMonitor as it becomes available. Information pertaining to errors is sent as they occur.

- **bool NodeMonitor::setAlgoInfo(MonData), (3)**
  This routine returns monitoring data to NodeMonitor in repsonse to a getAlgoInfo call.

- **bool NodeMonitor::sendError(ErrData), (4)**
  This routine delivers error information to NodeMonitor for delivery to T.E.D

6 Data Synchronization on the L2 Processors

6.1 Motivation

The NodeAlgo process on the L2 nodes will generate algorithm monitoring data that is read by the NodeMonitor logical processes. NodeMonitor packages this data and transmits it to the L2NM interface on T.E.D., as described in section 5. NodeAlgo will pass monitoring infomration to NodeMonitor through shared memory segments. Access to these shared regions must be synchronized so as to ensure the integrity of the data contained within. As mentioned in section 2.1, the process of monitoring the algorithms is required to introduce only minimal overhead to the operation of NodeAlgo. The mechanism used to establish data synchronization between NodeAlgo and NodeMonitor must also respect this constraint.

NodeAlgo and ModeMonitor represent logical tasks, *ie:* tasks that do not necessarily correspond directly with heavy-weight Linux processes or threads of execution. The function of NodeMonitor, for instance, may be performed by a process that also performs other tasks, such as that of NodeControl. The distinction between logical and actual processes is important for NodeMonitor but less so for NodeAlgo. It is clear that the high-priority operations of NodeAlgo are best handled by a thread(s) that is not encumbered by any other task. In the text that follows we describe a model for the synchronization of memory accesses between the two threads that implement NodeAlgo and NodeMonitor but the reader should bear in mind that the NodeMonitor thread may also be charged with additional responsibilities.

6.2 Reader & Writer Mutexes

Our preferred approach to the synchronization of data between NodeAlgo and NodeMonitor involves posix mutexes (mutual exclusion devices). Mutexes have two states, locked and unlocked, and can be used to allow single agent access to a protected resource. Locking and unlocking a mutex are "atomic" operations and guaruntee that a thread which has locked a mutex is the only thread to have done so. A more complete description of posix threads and mutexes may be found in Nichols, *et. al* [1].

Access to shared memory resources will be synchronized with two mutexes, as shown in Figure 3. In the course of performing the L2 trigger algorithms NodeAlgo will collect information on events at a specified rate. When NodeAlgo completes the collection of information for a given event it will lock the writer mutex (A) before writing the information to memory (B). The lock will temporarily prevent future access to the shared memory region by NodeAlgo while the already locked reader mutex prevents accesses by NodeMonitor. NodeAlgo finishes writing data and unlocks the reader mutex (C), permitting NodeMonitor to access the data. NodeMonitor is now able to lock the reader mutex (D) and read the monitoring data (E). When finished, NodeMonitor unlocks the writer mutex (F), granting NodeAlgo access to shared memory, and the process begins anew (G).

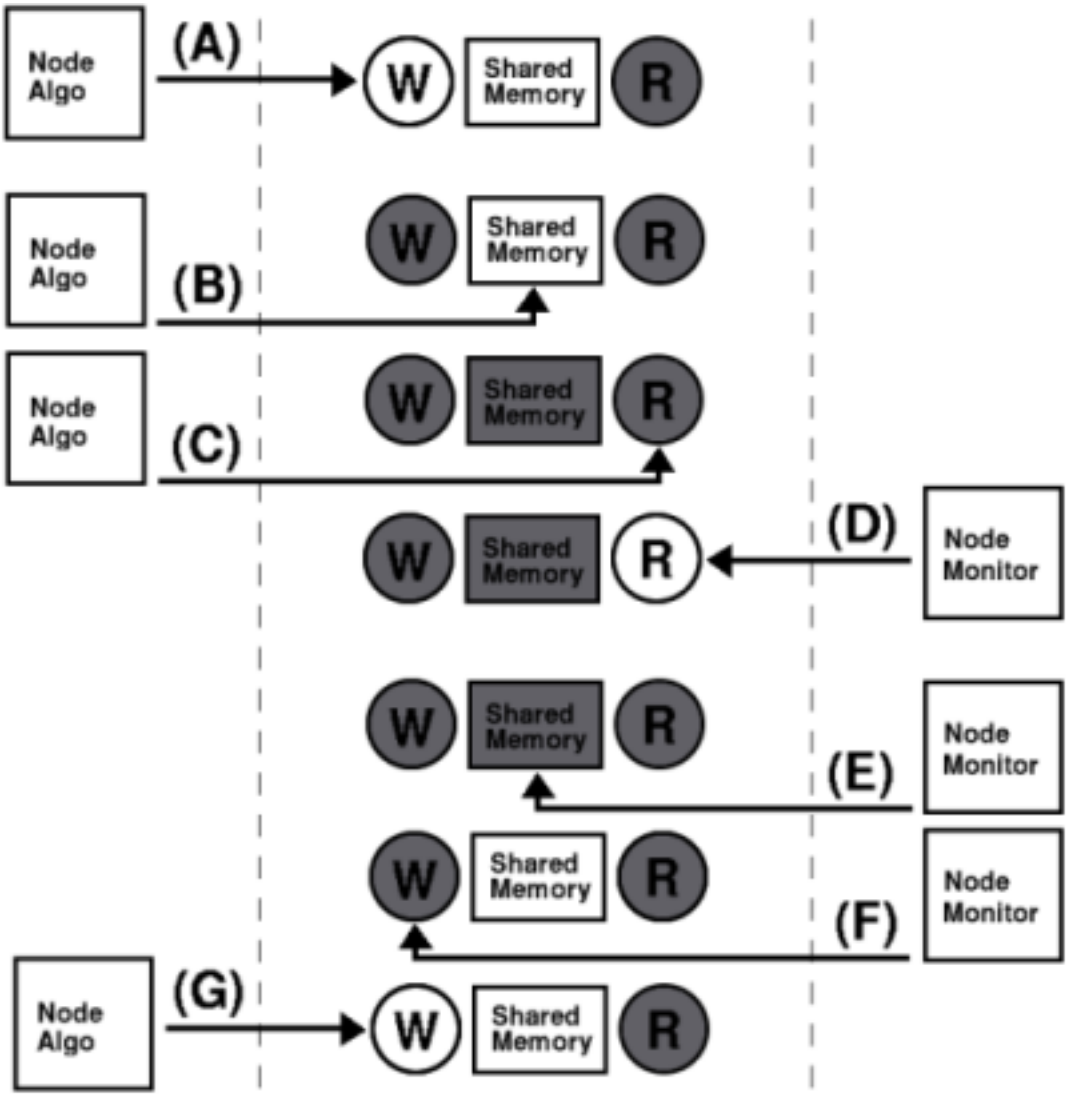

Figure 3: Access to memory shared between the NodeAlgo and NodeMonitor logical processes is synchronized using reader and writer mutexes, depicted here as the circles labeled "R" & "W". Locked mutxes and memory containing valid data are filled with gray. In the process shown NodeAlgo locks the writer mutex, writes monitoring information to shared memory and unlocks the reader mutex. NodeMonitor then locks the reader mutex, reads the data and unlocks the writer mutex, permitting NodeAlgo to access shared memory again.

Attempts by either thread to lock a mutex that it already holds will fail, indicating that the companion thread is accessing the shared data. NodeAlgo will try to obtain a lock on the writer mutex once before it goes on to process additional events. It will attempt to look the mutex again after completeing some specified number of events.

NodeMonitor's response to locking faliure remains as an implementation decision.

6.3 Timing

Meausurements of the time required to lock and unlock a mutex on a high-priority thread yield 1.7μs and 1.8μs, respectively. We performed tests using two mutexes and threads as described above. Details of the measurements are provided in a web page [2].

References

[1] the O'Reilley book

[2] http://www-cdf.fnal.gov/ krisitian/pulsar/sync.html